SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 8, 145–178 (1996)

Research

A Specification Driven Slicing Process for Identifying Reusable Functions

ANIELLO CIMITILE

Department of 'Ingegneria dell'Informazione ed Ingegneria Elettrica', University of Salerno, Faculty of Engineering at Benevento, 821000 Benevento, Italy

ANDREA DE LUCIA

Department of 'Informatica e Sistemistica', University of Naples 'Federico II', Via Claudio 21, 80125 Naples, Italy

MALCOLM MUNRO

Centre for Software Maintenance, University of Durham, South Road, DH1 3LE Durham, U.K.

SUMMARY

We present a new program slicing process for identifying and extracting code fragments implementing functional abstractions. The process is driven by the specification of the function to be isolated, given in terms of a precondition and a postcondition. Symbolic execution techniques are used to abstract the preconditions for the execution of program statements and predicates. The recovered conditions are then compared with the precondition and the postcondition of the functional abstraction. The statements whose preconditions are equivalent to the pre and postconditions of the specification are candidate to be the entry and exit points of the slice implementing the abstraction. Once the slicing criterion has been identified the slice is isolated using algorithms based on dependence graphs. The process has been specialized for programs written in the C language. Both symbolic execution and program slicing are performed by exploiting the Combined C Graph (CCG), a fine-grained dependence based program representation that can be used for most software maintenance tasks.

The work described in this paper is part of RE^2, a research project aiming to explore reverse engineering and re-engineering techniques for reusing software components from existing systems.

KEY WORDS: reverse engineering; re-engineering; reuse; program slicing; symbolic execution; logic specifications

1. INTRODUCTION

Code scavenging consists in searching existing software systems for source code components that implement software abstractions (Canfora, Cimitile and Visaggio, 1995). Software components recovered using code scavenging techniques can be re-engineered and clustered into different modules. There are many reasons for searching existing systems for software abstractions. Software evolution degrades the structure and increases the entropy of a software system (Lehman, 1984). After repeated changes, such a system

CCC 1040–550X/96/030145–34

Received 3 April 1995
Revised 22 December 1995

becomes 'legacy'. As an evolving system keeps its usefulness in the real world (Lehman, 1984), remedial actions are required to cope with it (Bennett, 1995).

A way to improve the understandability and maintainability of a legacy system is modularization. Modularizing an existing system consists of replacing a single large program or module with a functionally equivalent collection of smaller modules. Modularization using code scavenging techniques is also useful for downsizing large applications from mainframes to distributed client/server platforms (Sneed and Nyary, 1994). Indeed, as changing hardware platforms is becoming a question of vital importance for the economy and the competitiveness of many companies, software systems developed for the old platform have to be available on the new one. In many cases re-engineering the existing systems and adapting them to the new platform is cost-effective and can be preferable to new development (Sneed, 1995).

Another reason for identifying and extracting software components implementing abstractions in existing systems is for reusing them in the development of new software systems (Arnold and Frakes, 1992; Basili and Caldiera, 1991; Canfora, Cimitile and Munro, 1994). The benefits of software reuse in improving the productivity and the quality of new software projects and in reducing significantly the development costs (Lim, 1994) are faced with the effort needed for building up a repository of reusable components (Arnold and Frakes, 1992). A cost-effective way for obtaining reusable assets is by extracting and re-engineering them from existing software systems (Basili and Caldiera, 1991; Canfora, Cimitile and Munro, 1994).

Several code scavenging methods have been proposed in the literature (see Canfora, Cimitile and Visaggio (1995) for a survey). These methods can be grouped into three families: methods driven by a metric model (Fenton, 1991), methods driven by the type of the abstraction to be identified, and methods driven by a full or partial specification of the abstraction to be recovered. A metric driven method for the identification of reusable components has been defined, for example, by Basili and Caldiera (1991). Methods driven by the type of the abstraction to be recovered make use of structural reverse engineering techniques (Chikofsky and Cross, 1990) to extract a set of software components from code and make up instances of a predefined model of the abstraction's type. Several methods for the identification of different types of abstractions have been defined and used, for example, within the RE^2 project (Canfora, Cimitile and Munro, 1994). Methods driven by a metric model and by the type of the abstraction can also be called structural methods, because the recovered software components satisfy particular structural properties. Usually, a concept assignment process (Biggerstaff, Mitbander and Webster, 1994) has to be applied in order to associate the extracted components and the related relationshps with human orientated concepts. Such a process allows the selection of the set of meaningful software components (the components that actually implement software abstractions) among the set of the recovered components, for the following re-engineering phase. This is also useful to validate the code scavenging method (Cimitile, Fasolino and Maresca, 1993).

Specification driven methods presuppose the existence of information about the specification of the abstraction to be recovered. This information can be given in different ways. For example, a widely used approach is building up a library of programming *plans* (Soloway and Erdlich, 1984) or *clichés* (Rich and Waters, 1990) (each of which corresponds to an abstraction) and searching for their instances in code (Abd-El-Hafiz and Basili, 1993; Kontogiannis *et al.*, 1994; Kozaczynsky, Ning and Engberts, 1992, 1994;

Quilici, 1994; Wills, 1990, 1992). While methods based on plan libraries do not usually involve any execution model in order to recognise plans, other approaches use dynamic analysis techniques for mapping program features to code. For example, Wilde *et al.* (1992) and Wilde and Scully (1995) propose a method based on carefully designed test cases for locating code that implements specific user functionalities.

In this paper we present *specification driven program slicing* for the identification of code fragments implementing *functional abstraction.* Program slicing (Weiser, 1984) is a program decomposition technique useful for several software maintenance tasks (Gallagher and Lyle, 1991). An overview of program slicing techniques and their applications has been presented by Tip (1995). In particular program slicing has been used as a structural method for isolating functionalities in large programs (Canfora *et al.*, 1994b; Lanubile and Visaggio, 1993; Ning, Engberts and Kozaczynski, 1993). The identification of the code fragment implementing a functional abstraction cannot be achieved using pure program slicing, without the knowledge of the specification of the function to be recovered. Canfora *et al.* (1994a) use conditions of the specification for isolating a particular behaviour of the function. A recent work by Hall (1995) extends the theoretical framework of *dynamic program slicing* (Korel and Laski, 1990) for the extraction of a code fragment that correctly executes a given set of test cases. However, the main problem in using program slicing as a code scavenging technique is the setting up of a suitable slicing criterion for the extraction of a slice implementing a meaningful function. The slicing process we propose uses the specification of the functional abstraction to be isolated and symbolic execution (Clarke and Richardson, 1981; Coward, 1988; King, 1976) and theorem proving (Andrews, 1981; Boyer and Moore, 1979; Paulson, 1982) techniques to identify the slicing criterion. The specification of the function is given in terms of two first order logic formulas, called *precondition* and *postcondition* (Hoare, 1969). The process has been specialized for programs written in C language (Kernighan and Ritchie, 1988) and exploits the features of the Combined C Graph (CCG) (Kinloch and Munro, 1994), a fine-grained dependence based program representation.

The paper is organized as follows. In Section 2 the specification driven program slicing process is presented. Section 3 recalls the features of existing program representations and describes the Combined C Graph. In Section 4 symbolic execution is described, while Section 5 illustrates the technique used for finding slicing criteria and extracting program slices. Related work and concluding remarks are discussed in Sections 6 and 7, respectively.

2. SPECIFICATION DRIVEN PROGRAM SLICING

Program slicing has been introduced by Weiser (1981, 1984) as a powerful method for automatically decomposing a program by analysing its control and data flow. Program slices help programmers to better understand programs while debugging (Weiser, 1982) and can be used to parallelize sequential programs (Weiser, 1984). In Weiser's definition a *slicing criterion* of a program P is a pair $\langle s_{out}, V_{out} \rangle$ where s_{out} is a statement in P and V_{out} is a subset of variables in P. A *slice* of a program P on a slicing criterion $\langle s_{out}, V_{out} \rangle$ consists of all the statements and predicates of P that might affect the values of the variables in V_{out} just before the statement s_{out} is executed.

The program dependence graph (Ferrante, Ottenstein and Warren, 1987) can be used as a representation for implementing efficient slicing algorithms both at intraprocedural (Ottenstein and Ottenstein, 1984) and interprocedural (Horwitz, Reps and Binkley, 1990)

level. A slice is computed by backward traversing the control and data dependence edges of the graph and corresponds to the subgraph containing the reached vertices and edges. In order to exploit these algorithms, all the variables in the set V_{out} of a slicing criterion $\langle s_{out}, V_{out}\rangle$ are required to be used at s_{out}.

2.1. Identifying functional abstractions by program slicing

Program slicing has been used as structural method for isolating functionalities in large programs. For example, Ning, Engberts and Kozaczynski (1993) use program slicing to segment large legacy systems written in COBOL, while Canfora *et al.* (1994b) isolate the external functionalities of large monolithic programs. In general a structural method searches the code for software components that satisfy structural metrics (Basili and Caldiera, 1991) or make up an instance of a predefined model of abstraction based on particular relations between their subcomponents (Canfora, Cimitile and Visaggio, 1995). For example, a method for the identification of objects in code might search for sets of procedures and global variables in a program that respect *summary* relations built on the top of the *definition* and *use* relation between procedures and variables (Canfora, Cimitile and Munro, 1994).

A slicing based structural method for isolating functional abstractions can be defined with reference to Ottenstein's definition of slicing criterion. In this case the variables in the set V_{out} of the slicing criterion correspond to the output data of the function to be isolated and the software component identified (the slice) satisfy the transitive closure on the control and data-flow dependencies from the program point s_{out}. In the original definition a program slice is an executable code fragment starting from the entry of the program. Actually, such a code fragment cannot be always considered as a reusable function. Indeed, the use of program slicing for isolating reusable functions also involves the knowledge of the statements where the computation of the slice must end. For example, Lanubile and Visaggio (1993) provide information about the input data of the function to be isolated by adding a set of variables V_{in} to the slicing criterion. They also introduce the definition of the *transform slice* of a program P on a slicing criterion $\langle s_{out}, V_{in}, V_{out}\rangle$ as the set of statements and predicates of P that might affect the values of the variables in V_{out} just before the statement s_{out} is executed. The computation of a transform slice ends as soon as the definition of each of the variables in V_{in} is found. Therefore, a transform slice is only useful for the identification of a function whose input variables are not defined further on during its execution, but fails if the variables in V_{in} are redefined within the function.

A different approach can be followed if we consider that a function should have a unique entry point, i.e., a statement s_{in} which dominates all the other statements on the control flow graph (Aho, Sethi and Ullmann, 1986; Hecht, 1977).[1]

Definition 2.1. A flow graph $G = (s, N, E)$ is a directed graph where N is the set of nodes, $s \in N$ and E is the set of edges. For each node $n \in N$ there exists a path from s to n. Given two nodes n_1 and n_2 in N, n_1 dominates n_2 if and only if each path from s to n_2 contains n_1.

[1] In (Hecht, 1977), it is simply called a *flow graph*.

A new definition of slice can be obtained by adding such a statement s_{in} to the slicing criterion.

Definition 2.2. A slicing criterion of a program P is a triple $\langle s_{in}, s_{out}, V_{out} \rangle$ where s_{in} and s_{out} are statements in P, s_{in} dominates s_{out} and V_{out} is a subset of variables of P.

Definition 2.3. A slice of a program P on a slicing criterion $\langle s_{in}, s_{out}, V_{out} \rangle$ consists of all the statements and predicates of P that lie on a control flow path from s_{in} to s_{out} and that might affect the values of the variables in V_{out} just before the statement s_{out} is executed.

Cimitile and De Lucia (1995) have defined algorithms based on the control flow graph (Hecht, 1977) and the program dependence graph (Ferrante, Ottenstein and Warren, 1987) to compute such a slice.

2.2. Selecting a slicing criterion

Software components identified using a structural method need to be associated with a meaning. A concept assignment process (Biggerstaff, Mitbander and Webster, 1994) usually follows the code scavening phase: software components that can be associated with human orientated concepts will be re-engineered and reused, while meaningless software components are discarded (Cimitile, Fasolino and Maresca, 1993). The adequacy of a structural method can then be measured as the percentage of meaningful software components identified (Cimitile, Fasolino and Maresca, 1993).

Unfortunately, a pure structural method like program slicing is not completely adequate to isolate code fragments implementing functional abstractions. Indeed, program slicing might fail in the isolation of slices that can be associated with human orientated meanings, if a slicing criterion is not adequately selected. The most important problem in using program slicing for isolating reuse-candidate code fragments is then the identification of a suitable slicing criterion. This task requires the knowledge of the specification of the function we are looking for in code and cannot be completely automated.

The need to know the specification of the function to be isolated has been outlined by Canfora *et al.* (1994a). In this work a new slicing method, called *conditioned slicing*, has been defined in order to isolate slice fragments that can be executed whenever given conditions in the specification of the function hold true. However, in this work the specification of the function to be isolated is not used to identify the slicing criterion. In this paper we introduce a *specification driven program slicing* process for identifying a slice which implements a given specification of a functional abstraction. The specification of the functional abstraction is used together with symbolic execution (Clarke and Richardson, 1981; Coward, 1988; King, 1976) and theorem proving (Andrews, 1981; Boyer and Moore, 1979; Poulson, 1982) techniques in order to identify a suitable slicing criterion. We assume that the specification of a function is given in terms of its sets of *input* and *output* data and two first order logic formulas called *precondition* and *postcondition* (Hoare, 1969). The precondition expresses the constraint which must hold on the input data to allow the execution of the function, while the postcondition expresses the condition which will hold on the output and input data after the execution of the function (i.e., it relates the output data to the input data).

Figure 1 shows the specification driven program slicing process. From the source code of the program a *static analyser* produces a control flow based program representation.

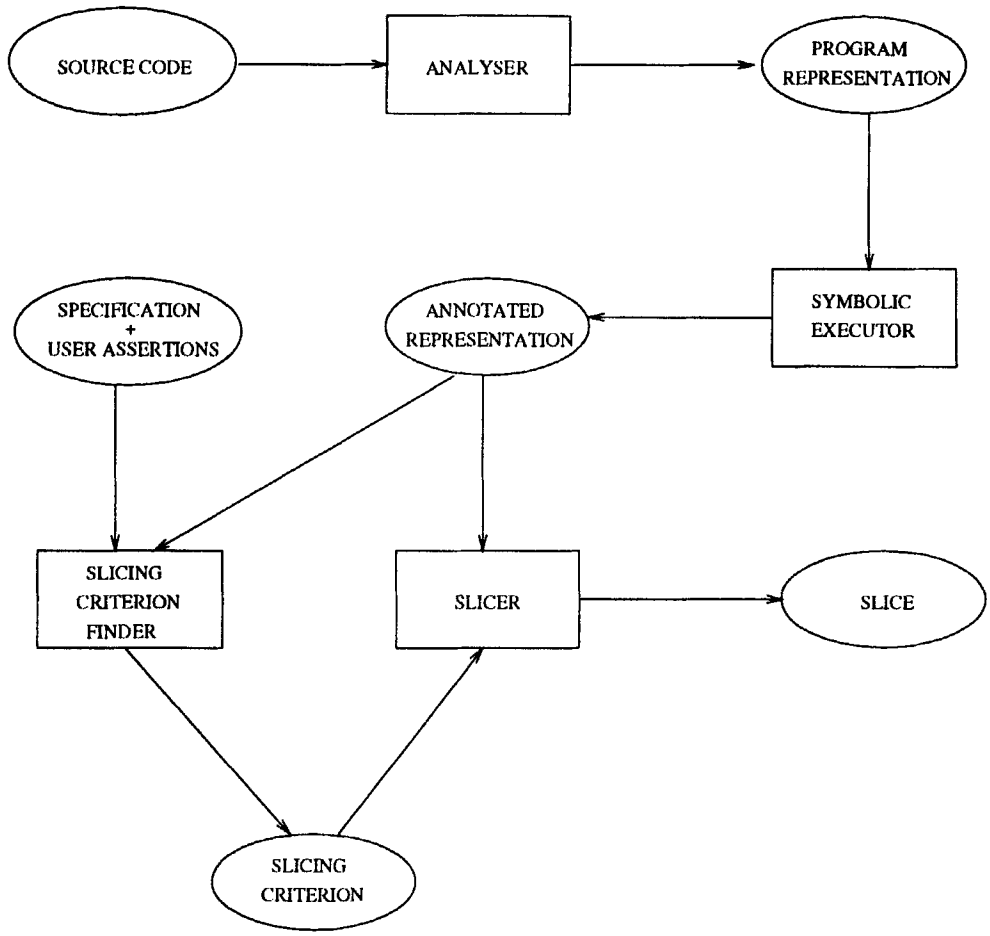


Figure 1. The specification driven program slicing process

A *symbolic executor* processes the program representation and associates each statement with the symbolic state holding before its symbolic exeuction (see Section 4). Such a symbolic state contains the precondition of the statement. The precondition of a statement is also called invariant assertion (see e.g. (Katz and Manna, 1976)), because it holds before the execution of the statement for each assignment to the symbolic constants. As the problem of finding invariant assertions is *undecidable*, human interaction is required to provide the assertions that cannot be automatically derived. Human interaction is also required to associate the data of the specification with the program variables and in particular to define the set of variables V_{out} corresponding to the output data of the function. These assertions and the specification of the function to be searched for are the input to the *slicing criterion finder*. Once a statement has been annotated with its entry symbolic state, the finder checks the equivalence of the statement precondition with the precondition and postcondition of the specification. A statement whose precondition is equivalent to the input precondition is candidate to be the statement s_{in} of the slicing criterion, while a statement whose precondition is equivalent to the input postcondition is

candidate to be the statement s_{out} of the slicing criterion. If s_{in} also dominates s_{out} the slicing criterion $\langle s_{in}, s_{out}, V_{out} \rangle$ is produced as output. Finally, a *program slicer* computes the slice and isolates the code implementing the functional abstraction. In the remaining sections we show a specialization of this process to programs written in the C programming language (Kernighan and Ritchie, 1988).

3. A PROGRAM REPRESENTATION FOR C PROGRAMS

The specification driven program slicing process requires a program representation able to provide a maintainer with different views of a program. Algorithms such as those to calculate data flow information and program slices involve the use of intermediate graph-based program representations, while symbolic execution requires the same syntactic and semantic information (often represented by annotated abstract syntax trees) as compilers and interpreters (Aho, Sethi and Ullmann, 1986). The intermediate program representation used in the specification driven program slicing process for C programs is the Combined C Graph (CCG) introduced by Kinloch and Munro (1994). The CCG extends the work by Harrold and Malloy (1993) for combining the features of different program representations into a single unified intermediate representation for a maintenance environment. In this section we will recall some of the existing program representations and describe the CCG.

3.1. Existing program representations

The simplest program representations used in software maintenance are the *control flow graph* (Aho, Sethi and Ullmann, 1986; Hecht, 1977) and the *call graph*. A control flow graph consists of nodes that represent single-entry/single-exit regions of executable code (called *basic blocks*) and edges which represent possible execution flow between code regions. Although for compiler optimization (Aho, Sethi and Ullmann, 1986) a basic block consists of a maximal single-entry/single-exit sequence of statements, usually in software maintenance each node of a control flow graph corresponds to an individual statement. A control flow graph is suitable to perform intraprocedural data flow analysis (Aho, Sethi and Ullmann, 1986; Hecht, 1977) and program slicing (Weiser, 1984). In a call graph, nodes represent procedures, edges represent call relationships between procedures, and edges' labels represent actual parameters. Since one procedure may call another at many points, a call graph may be a multigraph with more than one edge connecting two nodes. A program's call graph can be constructed efficiently (Ryder, 1979) and used for different applications, like *flow-insensitive* interprocedural data flow analysis, e.g. (Cooper and Kennedy, 1989; Torczan and Kennedy, 1984), and software salvaging (Cimitile, Fasolino and Maresca, 1993; Cimilile and Visaggio, 1995).

The features of these two representations have been combined in the *interprocedural control flow graph* (Landi and Ryder, 1991). An interprocedural control flow graph is the union of the control flow graphs of the individual procedures in the program; each control flow graph has unique *entry* and *exit* nodes and *call sites* are split into *call* and *return* nodes. Each call node is connected to the entry node of the procedure it invokes, while each exit node is connected to return nodes of all call sites that invoke the procedure. The interprocedural control flow graph has been used to solve the *reaching definitions* problem (i.e., the problem of determining the set of variable definitions that reach each

program point) in the presence of points (Pande, Landi and Ryder, 1994). A similar representation has been used by Myers (1981) for flow-sensitive interprocedural data flow analysis.

A variant of the call graph which provides information for flow-sensitive data flow analysis is the *program summary graph* (Callahan, 1988). The program summary graph represents programs written in a procedural language with call by reference parameters. For each procedure, there are *entry* and *exit* nodes for each formal parameter, while at each call site there are *call* and *return* nodes for each actual parameter. Binding edges relate call nodes with the corresponding entry nodes and exit nodes with the corresponding return nodes. Data flow within each procedure is summarized by *reaching edges* between the procedure control points such as entry, exit, call and return for formal and actual parameters. Callahan (1988) presents iterative algorithms to solve a variety of data flow problems such as whether the values of actual reference parameters *may be preserved* over a procedure call. The program summary graph and the algorithms that use it for flow-sensitive data flow analysis were developed to re-engineer existing Fortran programs for parallel environments.

An extension of the program summary graph, the *interprocedural flow graph* (Harrold and Soffa, 1990) allows the calculation of interprocedural definition-use pairs (Aho, Sethi and Ullmann, 1986), by providing information at each node about the locations of definitions and uses of reference parameters and global variables that can be reached across procedure boundaries. Two algorithms compute the *reaching definitions* and *reachable uses* (i.e., the problem of determining the set of variable uses that can be reached from each program point), by propagating the intraprocedural data flow information throughout the program guided by the edges in the graph. To ensure that data flow information is propagated only over possible execution paths in the program, new edges connecting call and return nodes (*interprocedural reaching edges*) are used to preserve the calling context of the called procedures during the interprocedural data flow analysis. The resulting sets of reaching definitions and reachable uses can be used to compute definition-use pairs for integration testing (Harrold and Soffa, 1991).

The *program dependence graph* (Ferrante, Ottenstein and Warren, 1987) is an intraprocedural program representation consisting of two superimposed subgraphs, the *control dependence subgraph* and the *data dependence subgraph*. In the control dependence subgraph, nodes represent program statements and edges represent control dependencies[2] between statements. The data dependence subgraph contains several types of edges representing different data dependence relations (Kuck, 1978), including definition-use pairs (Aho, Sethi and Ullmann, 1986). It was first introduced as an internal representation for optimizing and parallelizing compilers (Ferrante, Ottenstein and Warren, 1987). Its role in a software development and maintenance environment has been outlined by Ottenstein and Ottenstein (1984).

An extension of the program dependence graph, the *system dependence graph* (Horwitz, Reps and Binkley, 1990) combines dependence graphs for individual procedures with additional nodes and edges making up the call interfaces. Nodes are added to a procedure's dependence graph to model parameter passing by value-result. Each call-site node is

[2] Given a control flow graph of a program, a node *m* is *control dependent* on a node *n* if and only if *n* has two outgoing edges, where one of them always results in *m* being reached, while the other edge may result in *m* not being reached (Ferrante, Ottenstein and Warren, 1987).

connected to the entry node of the called procedure. Moreover, the call-site contains *actual-in* and *actual-out* nodes for each actual parameter, while *formal-in* and *formal-out* nodes are linked to the procedure's entry node for each formal parameter. Binding edges connect actual-in with formal-in nodes and formal-out with actual-out nodes. The system dependence graph has been introduced by Horwitz, Reps and Binkley (1990) for interprocedural slicing. The original interprocedural slicing algorithm proposed by Weiser (1984) suffers lack of precision, because it fails to account for the calling context of a called procedure. *Interprocedural data flow edges* modelling transitive data dependencies between actual parameters across a procedure call are then added to the system dependence graph and exploited, in order to preserve the procedure calling context during interprocedural slicing. These edges permit a more precise computation of a slice across procedure boundaries. Livadas and Croll (1994) extend the statement based system dependence graph to support both pass-by-value and pass-by-reference parameters. Moreover, any number of C like return statements are allowed to appear anywhere in a function. A parse tree is used as the basis of the system dependence graph. This allows slicing to be more precise than if *resolution* of the system dependence graph was only at statement level.

Harrold and Malloy (1993) identified the problem that a maintenance environment needs information contained in different intermediate program representations. Instead of incorporating each of the existing representations and using the associated algorithms to develop program maintenance tools, they propose to use the *unified interprocedural graph*, an interprocedural program representation that integrates the features of four different representations: the call graph, the program summary graph, the interprocedural flow graph and the system dependence graph. Algorithms developed for each of these program representations are applicable to the unified interprocedural graph by simply considering subsets of nodes and edges. The main benefits of this approach are the reduction in storage space, deriving from the elimination of redundant information contained in the different representations, and the convenience of accessing a single program representation. Moreover, the construction of the unified interprocedural graph can be obtained incrementally.

A similar approach has been followed in the Web structure orientated Software Development Workbench (WSDW) (De Lucia *et al.*, 1992). In WSDW, software development and maintenance tools are integrated through sharing the same program representation, called *web structure* (Maggiolo Schettini, Napoli and Tortora, 1988). A web structure is a particular *relational structure* (Ehrig *et al.*, 1981) originally designed as internal program representation for compilers and interpreters. It consists of a supporting tree structure with labelled nodes which represents the syntax of the program. A second structure is superimposed which consists of labelled edges, called *web edges*, connecting nodes of the underlying tree. Web edges provide semantic information, e.g., about control flow, call interface, scope of variables, and other properties that the tree alone cannot express. Program analysis and transformations are accomplished by rewriting the web representation according to rewriting rules called productions (Maggiolo Schettini, Napoli and Tortora, 1988). Tools based on the algebraic framework of web transformations can enrich the representation with new information or transform the program structure. For example, in WSDW (De Lucia *et al.*, 1992) a *data dependency analyser* adds data dependence edges (Kuck, 1978) to the web representation, while a *program recoder* is used for program restructuring and a *program parallelizer* transforms the web representation of a sequential program into the web representation of an equivalent parallel program.

3.2. The Combined C Graph

The CCG combines all the program representations described in the above sections (in particular the interprocedural control flow graph, the unified interprocedural graph and the web structure) and considers a variety of language constructs such as pointer and structure variables, expressions with embedded side-effects or control flows and value-returning functions. The CCG is composed of a collection of Function CCGs (FCCGs) supporting both syntactic and semantic information of an individual C function. A control flow graph and a variety of other edge types, including data and control dependence edges, are superimposed to an abstract syntax tree (Cimitile, De Lucia and Munro, 1995a). The FCCGs are connected by various interprocedural edges, like call interface edges (enclosing binding edges between actual and formal parameters and between return statements and calling sites) and interprocedural data dependencies.

The main feature of the CCG is the ability to represent embedded side-effects and control flows contained in a C subexpression. Side-effects can arise as a result of assignment operators, increment and decrement operators, comma operator and function calls, while embedded changes in the control flow can arise because of the *short-circuit* evaluation of the boolean expressions (Kernighan and Ritchie, 1988). Embedded side-effects and control flows may give rise to data or control dependencies between individual subexpressions. Therefore, more accurate program slices can be produced by taking into account this information (see Section 5). Rather than a one to one correspondence between the function's statements and the vertices of the control flow subgraph in its FCCG, a finer-grained representation of these statements is required to deal with embedded side-effects and control flows. Whenever a statement contains embedded side-effects or control flows an additional vertex is created for each subexpression containing a definition or a possible change in the control flow. For example, for an assignment expression:

- a = b

the following vertices are created:

- (a = b) : no side-effects
- (a, (= b) : side-effect in a
- (b), (a =) : side-effect in b
- (a), (b), (=) : side-effect in a and b

We assume that the control flow between these vertices is from left to right. In this way, the vertex representing the complete statement is always the last of the sequence. Analogously, a boolean expression with short-circuiting evaluation:

- a && b || c

generates the three vertices

- (a), (&& b), (|| c)

and the internal control flow edges (labelled *true* and *false*, respectively):

- (a) $\rightarrow_{\text{true}}$ (&& b)
- (a) $\rightarrow_{\text{false}}$ (|| c)
- (&& b) $\rightarrow_{\text{false}}$ (|| c)

while a conditional expression:

- a ? b : c

gives the three vertices:

- (a), (b), (c)

and the control flow edges:

- (a) $\rightarrow_{\text{true}}$ (b)
- (a) $\rightarrow_{\text{false}}$ (c)

Two edge types are used to relate the extra vertices of an expression, *expression-use edges* and *lvalue definition edges*. An expression-use edge from vertex p to vertex q, $p \rightarrow_{\text{eu}} q$ indicates the evaluation of an expression at vertex p followed by a use of the resulting value at q. For example, this situation occurs when the expression in the right-hand side of an assignment contains a side-effect or an embedded control flow. An *lvalue* (left value) is an expression referring to a named region of storage in the left-hand side of an assignment. An lvalue definition edge from vertex p to vertex q, $p \rightarrow_{\text{ld}} q$ indicates the evaluation of an lvalue at vertex p followed by a writing to the corresponding storage location at vertex q. This situation occurs when the operand in the left-hand side of an assignment contains a side-effect. For example, the expression:

- *p++ = x = 0

generates the three vertices:

- (*p++), (x = 0), (=)

where the last vertex corresponds to the left-most assignment, and the two special edges

- (x = 0) $\rightarrow_{\text{eu}}$ (=)
- (*p++) $\rightarrow_{\text{ld}}$ (=)

Other intraprocedural edges are used to represent data dependencies, in particular dataflow dependencies, and control dependencies between the vertices of the control flow subgraph. Interprocedural edges concerns

- *call* edges from a call vertex in the calling function to the entry vertex of the called function;
- *parameter binding* edges from each actual parameter vertex to its corresponding formal parameter vertex;

- *return expression-use* edges from a return statement vertex in the called function to the call vertex in the calling function;
- interprocedural data dependence edges.

A more complete discussion is contained in Kinloch and Munro (1994). As an example, let us consider the following simple program

```
main() {
  int a, b;
  a = 1 + (b = 0);
  while(a <= 10)
    b += double(&a);
}

int double(int *p) {
      return (*p)++ * 2;
}
```

Figure 2 shows the CCG representation of the program above. For the sake of simplicity, variable and formal parameter declarations and variable references in the syntax tree are represented by the corresponding identifiers, without semantic information. Figure 2(a) shows the abstract syntax tree and some of the superimposed edge types. Control and data flow dependence edges are depicted in Figure 2(b).

4. SYMBOLIC EXECUTION

Symbolic execution was first introduced as a powerful tool for program testing (Clarke and Richardson, 1981; King, 1976) and program verification (Dannenberg and Ernst, 1982; Hentler and King, 1976). More recently, it has been used for software specialization (Coen-Porisini *et al.*, 1991) and for recovering the formal specifications of reusable software components (Abd-El-Hafiz, Basili and Caldiera, 1991; Cimitile, De Lucia and Munro, 1995a).

The traditional execution model of a program is based on the control flow and on the concept of state. A program state is a set of pairs $\{\langle M_1, v_1\rangle, \langle M_2, v_2\rangle, \ldots, \langle M_n, v_n\rangle\}$, where $M_1, M_2, \ldots, M_n$ are the memory locations corresponding to the program's variables and for $1 \leq i \leq n$, v_i is the value stored in M_i at some point of the execution. An execution of a program can be represented by a sequence $S_0 p_1 S_1 \ldots p_f S_f$, where each S_i, $0 \leq i \leq f$, is a program state and each p_j, $1 \leq j \leq f$ is a program statement or predicate. S_0 is the initial state and S_i, $1 \leq i \leq f$, is the state resulting from the execution of p_i in the state S_{i-1}; S_f is the final state. If p_j, $1 \leq j \leq f$, is a predicate, the information contained in S_{j-1} unequivocally allows the selection of the branch to follow. Moreover, the resulting state S_j will not change (with respect to the state S_{j-1}) if p_j does not contain side-effects.

While in traditional execution the values of program variables are constants, in symbolic execution they are represented by symbolic expressions, i.e., expressions containing symbolic constants. For example, the value v of a variable x might be represented by '$2*\alpha + \beta$', where α and β are symbolic constants. Like a traditional execution, a symbolic execution of a program might be represented by a sequence of alternating states and

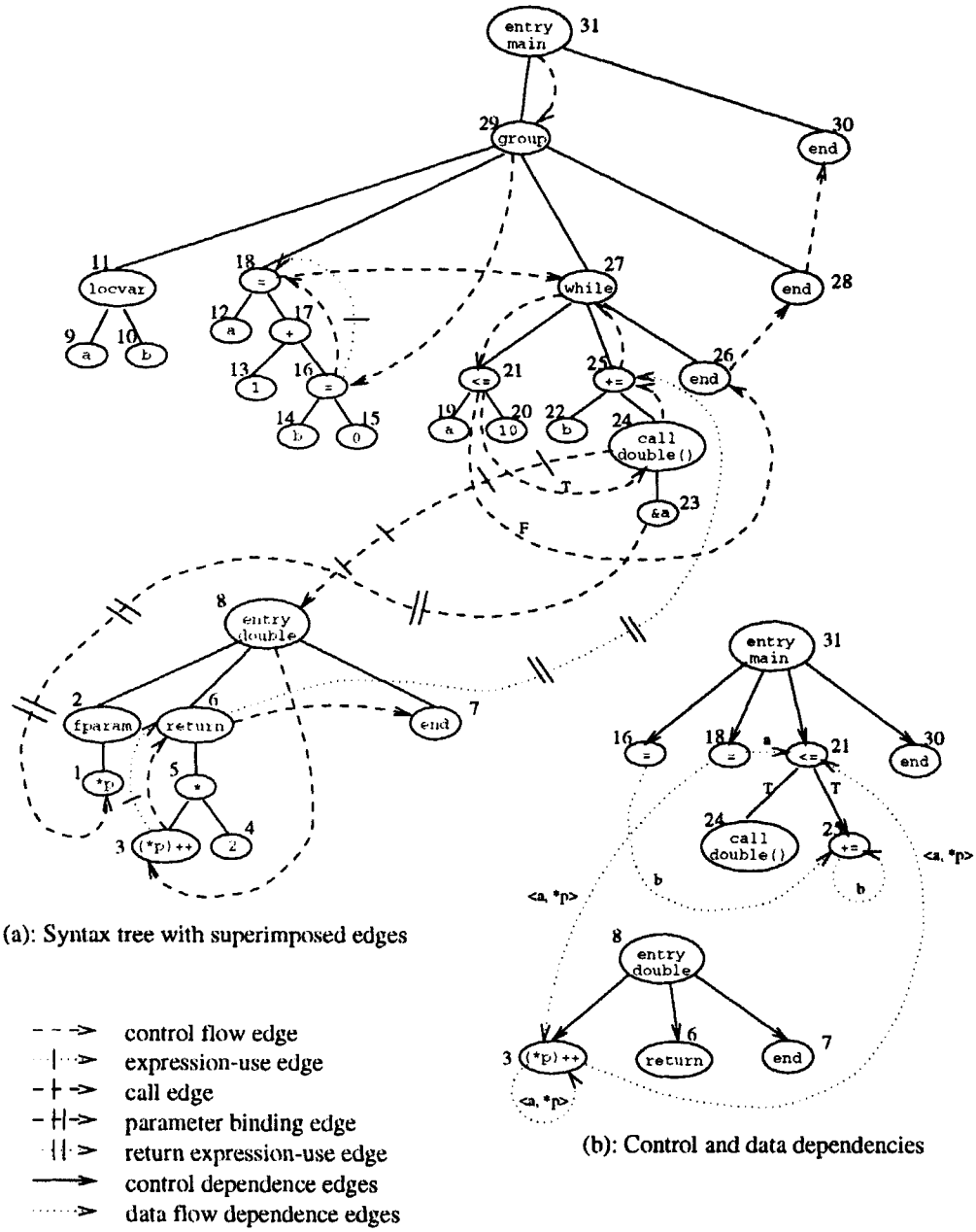


Figure 2. Example CCG

statements. However, if p_j is a predicate, the information contained in the state S_{j-1} may not suffice to select the branch to follow. Indeed, the symbolic boolean expression obtained by replacing the variables in p_j with the corresponding values in S_{j-1} could hold true for some assignments to the symbolic constants and false for other ones. For example, let us consider the following simple function:

```
int absdiff(int a, int b)
{
  int diff;
  if(a > b)
    diff = a - b;
  else
    diff = b - a;
  return diff;
}
```

and the evaluation of the predicate a > b in the state:

$$\{\langle a, \alpha\rangle, \langle b, \beta\rangle, \langle diff, \mathit{undef}\rangle\}$$

where α and β are the symbolic input values of a and b, respectively, and *undef* stands for an undefined value. The *true* branch has to be taken for each assignment to the symbolic constants α and β such that $\alpha > \beta$, otherwise the *false* branch has to be taken. In the following, the symbols $\wedge$, $\vee$ and $\neg$ will denote the logical *and*, *or* and *not*, respectively.

Whatever the selected branch is we must keep track of the condition which caused the branch to be selected, i.e., if $p_j(S_{j-1})$ is the symbolic boolean expression resulting from the evaluation of the predicate p_j in the state S_{j-1}, we must associate with the resulting state S_j the expression $p_j(S_{j-1})$, if the *true* branch has been selected, the expression $\neg p_j(S_{j-1})$, otherwise. Therefore, a symbolic state is a pair $\langle State, PC\rangle$, where *State* is as usual a set of pairs of the form $\langle M, \alpha\rangle$, M and α being a memory location and a symbolic expression respectively, and PC is a first order logic formula called *path-condition* and representing the condition which must be satisfied in order for an execution to follow the particular associated path on the control flow. For example, the evaluation of the predicate above a > b in the symbolic state:

$$\langle S_1, P_1\rangle = \langle\{\langle a, \alpha\rangle, \langle b, \beta\rangle, \langle diff, \mathit{undef}\rangle\}, \mathit{true}\rangle$$

generates two possible independent executions depending on the selected branch. The resulting symbolic state will be:

$$\langle S_2, P_2\rangle = \langle\{\langle a, \alpha\rangle, \langle b, \beta\rangle, \langle diff, \mathit{undef}\rangle\}, \alpha - \beta > 0\rangle$$

whenever the *true* branch is selected,

$$\langle S_3, P_3\rangle = \langle\{\langle a, \alpha\rangle, \langle b, \beta\rangle, \langle diff, \mathit{undef}\rangle\}, \beta - \alpha \geqslant 0\rangle$$

otherwise. On the contrary, if the predicate a > b is symbolically executed in the symbolic state:

$$\langle\{\langle a, \alpha\rangle, \langle b, \beta\rangle, \langle diff, \mathit{undef}\rangle\}, \alpha = \beta + \gamma \wedge \gamma > 0\rangle$$

only the *true* branch can be selected and the path-condition of the resulting symbolic state will remain unchanged (because $\alpha = \beta+\gamma \wedge \gamma>0 \wedge \alpha-\beta>0$ is equivalent to $\alpha = \beta+\gamma \wedge \gamma>0$). Indeed, the path-condition of the symbolic state resulting from the selection of the *false* branch would be $\alpha = \beta+\gamma \wedge \gamma>0 \wedge \beta-\alpha\geq 0$ which is identically false.

The previous example shows that when a predicate p is encountered in the symbolic state $\langle S, PC\rangle$, at most one of the following two implications can hold true (if PC is not identically false):

(a) $PC \Rightarrow p(S)$
(b) $PC \Rightarrow \neg p(S)$

If exactly one implication holds true, the execution continues on the *true* branch, when the implication (a) holds true or on the *false* branch, when the implication (b) holds true. In both cases the path-condition PC does not change. However, very often neither the implication (a) nor the implication (b) holds true, because there exists at least one set of inputs to the program satisfying the implication (a) and another set satisfying the implication (b). In this case, both the *true* and the *false* branches must be selected and two symbolic executions will proceed independently. Whenever the *true* branch is selected, $p(S)$ must hold true. This information has to be recorded in the path-condition which changes in $PC \wedge p(S)$. Analogously, whenever the *false* branch is selected, the path-condition becomes $PC \wedge \neg p(S)$. Although the problem of proving the implications (a) and (b) is in general *undecidable*, in practical cases an automatic theorem prover (Andrews, 1981; Boyer and Moore, 1979; Paulson, 1982) can be used. However, human interaction is required to make some decisions whenever the theorem prover is not able to reach a result.

The presence of predicates in a program and their dependence on the program's input generates multiple symbolic executions of a program (King, 1976). However, two independent symbolic executions could join at a program statement. For example, let us consider again the function above. As we have seen, the evaluation of the predicate a > b in the symbolic state $\langle S_1, P_1\rangle$ generates two independent executions. Whenever the *true* branch is selected, the statement diff = a–b is executed in the symbolic state $\langle S_2, P_2\rangle$ producing the symbolic state:

$$\langle S_4, P_4\rangle = \langle\{\langle a, \alpha\rangle, \langle b, \beta\rangle, \langle diff, \alpha - \beta\rangle\}, \alpha - \beta > 0\rangle$$

The selection of the *false* branch will generate the execution of the statement diff = b – a in the symbolic state: $\langle S_3, P_3\rangle$ producing the symbolic state

$$\langle S_5, P_5\rangle = \langle\{\langle a, \alpha\rangle, \langle b, \beta\rangle, \langle diff, \beta - \alpha\rangle\}, \beta - \alpha \geq 0\rangle$$

At this point, the two symbolic executions can be joined in a single execution by collapsing the two symbolic states. The *state folding* operator (Coen-Porisini *et al.*, 1991)

takes as arguments two symbolic states and produces a new symbolic state resulting from the composition of its arguments. If a variable has the same value in both the input symbolic states, its value remains unchanged in the resulting state, otherwise it receives a new symbolic value. The path-condition of the resulting state is the conjunction of two parts corresponding to the two input symbolic states. Each part is the conjunction of the path-condition and a predicate binding the new symbolic values to the symbolic expressions the corresponding variables have in the symbolic state. For example, the composition of the symbolic states $\langle S_4, P_4\rangle$ and $\langle S_5, P_5\rangle$ by the state folding operator before the symbolic execution of the statement return diff will result in the state:

$$\langle S_6, P_6\rangle = \langle\{\langle a, \alpha\rangle, \langle b, \beta\rangle, \langle diff, \gamma\rangle\}, (\alpha - \beta > 0 \wedge \gamma = \alpha - \beta) \vee (\beta - \alpha \geq 0 \wedge \gamma = \beta - \alpha)\rangle$$

The memory locations a and b have the same symbolic values in both S_4 and S_5 and then their values remain unchanged in S_6. On the contrary, the memory location diff has different values in S_4 and S_5 ($\alpha - \beta$ and $\beta - \alpha$, respectively) and receives a new symbolic value γ in S_6. The path-condition P_6 must consider both the joined symbolic executions and is the disjunction of two parts. The first part is the conjunction of the path-condition P_4 (i.e., $\alpha - \beta > 0$) with a predicate binding the values the memory location diff has in the states P_6 (i.e., γ) and P_4 (i.e., $\alpha - \beta$). Analogously, the second part is the conjunction of the path-condition P_5 (i.e., $\beta - \alpha \geq 0$) with a predicate binding the values the memory location diff has in the states P_6 (i.e., γ) and P_5 (i.e., $\beta - \alpha$).

4.1. Symbolic execution of C programs

The main problems with symbolic execution of C programs are pointer variables and expressions containing embedded side-effects and control flows (Cimitile, De Lucia and Munro, 1995a). To deal with alias variables each memory location is associated with its symbolic address and with the type of the stored information. Hence, in a symbolic state:

$$\langle\{\langle M_1, \alpha_1\rangle, \langle M_2\rangle, \alpha_2, \ldots, \langle M_n, \alpha_n\rangle\}, PC\rangle$$

each memory location M_i, $1 \leq i \leq n$ is a pair $\langle \mu_i, T_i\rangle$ where μ_i and T_i are the symbolic address and the type, respectively, of M_i.[3] Moreover, given a symbolic address μ, the expression $p(\mu)$ is meant to refer to the memory location having μ as symbolic address. For example, the declaration:

```
int x[3], *p;
```

will generate four memory locations:[4]

x[0] = $\langle \chi$, int$\rangle$
x[1] = $\langle \chi + 2$, int$\rangle$
x[2] = $\langle \chi + 4$, int$\rangle$
p = $\langle \pi$, *int$\rangle$

[3] We consider the memory as being an array of bytes. As a consequence, an address is any of the array positions.
[4] We assume that two bytes are required to store an integer value.

In general, the symbolic address of an array element $X[i]$ of type T is:

$$\chi + \text{sizeof}(T)*i$$

where χ is the symbolic address of $X[0]$ and the operator *sizeof* returns the number of bytes required to store a value of type T. The extension to the case of multidimensional arrays can be easily obtained.

Let us suppose to symbolically execute the following program fragment:

```
p = x;
*p = *p + 1;
p++;
```

in the symbolic state:

$$\langle\{\langle x[0],\ \alpha_1\rangle,\ \langle x[1],\ \alpha_2\rangle,\ \langle x[2],\ \alpha_3\rangle,\ \langle p,\ \textit{undef}\rangle\},\ PC\rangle$$

The assignment p = x gives rise to the aliases *p and x[0]. Its symbolic execution will result in the assignment of the symbolic address of x[0] to the memory location p producing the state:

$$\langle\{\langle x[0],\ \alpha_1\rangle,\ \langle x[1],\ \alpha_2\rangle,\ \langle x[2],\ \alpha_3\rangle,\ \langle p,\ \chi\rangle\},\ PC\rangle$$

As the symbolic value of p is χ, the expression *p in the statement *p = *p + 1 will refer to the memory location $p(\chi)$, i.e., the location x[0]. The state resulting from the symbolic execution of this statement is:

$$\langle\{\langle x[0],\ \alpha_1 + 1\rangle,\ \langle x[1],\ \alpha_2\rangle,\ \langle x[2],\ \alpha_3\rangle,\ \langle p,\ \chi\rangle\},\ PC\rangle$$

Finally, the symbolic execution of the statement p++ will produce the state:

$$\langle\{\langle x[0],\ \alpha_1 + 1\rangle,\ \langle x[1],\ \alpha_2\rangle,\ \langle x[2],\ \alpha_3\rangle,\ \langle p,\ \chi + 2\rangle\},\ PC\rangle$$

where the symbolic value of p is obtained as:

$$\chi + 1*\textit{sizeof}(\textit{int}) = \chi + 2$$

In general, if the type and the symbolic value of a memory location M are $*T$ (i.e., M is the memory location of a pointer variable) and α (α is the symbolic address of a memory location), respectively, the symbolic value of the expression $M+i$ will be:

$$\alpha + i*\textit{sizeof}(T)$$

If such a value is generated during a symbolic execution it should coincide with the symbolic address of a memory location M' in the current symbolic state. However, it is

possible that this value does not coincide with any symbolic address.[5] In this case a new memory location $\langle \alpha+i*\text{sizeof}(T), T \rangle$ is created and added to the symbolic state.

Embedded side-effects and control flows also play a critical role in symbolic execution of C programs. Embedded side-effects involve a state change for each variable definition and short-circuit evaluation of a boolean expression might affect the path-condition and create more intermediate independent executions. Hence, while the evaluation order of the operands in a commutative binary operation should not affect the final result, it plays a critical role if the operands may contain embedded side-effects or control flows. For example, let us consider the symbolic execution of the following *if* statement:

$$\text{if (a} > \text{b} \,||\, \text{a} > \text{c++ \{ ... \}}$$

in the symbolic state $\langle \{\langle a, \alpha \rangle, \langle b, \beta \rangle, \langle c, \gamma \rangle\}, PC \rangle$. Whenever $PC \Rightarrow \alpha > \beta$ the right-hand side operand of the || operator is not evaluated and the variable c is not incremented, while if $PC \Rightarrow \alpha \leqslant \beta$ a change in the symbolic state will result because of the increment operator.

Using a fine-grained program representation like the CCG a symbolic executor can easily deal with such problems. Symbolic execution is performed by traversing the control flow subgraph with tokens that contain the symbolic state corresponding to the followed execution path (Cimitile, De Lucia and Munro, 1995a). If ambiguous expressions like:

$$\text{*p++ = ++b + a-- + b}$$

are not allowed,[6] the symbolic executor can evaluate the subexpressions containing embedded side-effects, whenever the corresponding vertices are encountered on the control flow. For example, let us consider the following statement:

$$\text{c += a + b++;}$$

where a, b and c are integers variables. The CCG control flow subgraph for this statement is depicted in Figure 3. The symbolic execution of such a statement in the symbolic state:

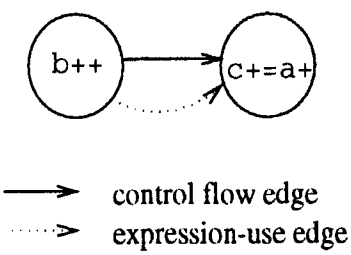


Figure 3. Example CCG subgraph

[5] In C pointers can be used to simulate arrays without fixed length or dynamic structures.

[6] With the exception of the booelan operators && and ||, the conditional operator ?: and the comma operator, the order of evaluation for operands within C expressions is undefined (Kernighan and Ritchie, 1988). If α and β are the values of the variable a and b before the execution of the statement *p++ = ++b + a-- + b, a left to right evaluation of the right-hand side of the assignment expression gives $\alpha + 2*\beta + 2$ as result, while a right to left evaluation gives $\alpha + 2*\beta + 1$. However, compilers impose an evaluation order for operands within expressions and usually they allow ambiguous expressions.

$$\langle\{\langle a, \alpha\rangle, \langle b, \beta\rangle, \langle c, \gamma\rangle\}, PC\rangle$$

is performed in the following steps:

- symbolic execution of the expression b++: evaluate the value stored in b and increment it; the evaluation result is: val(b++) = β,[7] while the new state is

$$\langle\{\langle a, \alpha\rangle, \langle b, \beta{+}1\rangle, \langle c, \gamma\rangle\}, PC\rangle$$

- symbolic execution of the expression c += a + val(b++):[8] evaluate the expression:

$$c + a + val(b{+}{+}) = \gamma + \alpha + \beta$$

and assign it to c; the resulting state will be:

$$\langle\{\langle a, \alpha\rangle, \langle b, \beta + 1\rangle, \langle c, \alpha + \beta + \gamma\rangle\}, PC\rangle$$

If ambiguous expressions are allowed, we must decide an evaluation order for the expressions. However, in this case we cannot take advantage of the CCG extra-vertices representing embedded side-effects. Indeed, nodes representing embedded side-effects of an expression must be skipped until the node representing the whole expression statement is reached (a node representing an embedded side-effect can be recognized because of its outgoing expression-use or lvalue definition edge). Whenever the token reaches a statement node the symbolic evaluation of the corresponding expression starts with respect to the symbolic state associated with the token. The symbolic evaluation is made by traversing the abstract syntax tree of the expression according to the chosen evaluation order. Whenever a subexpression contains a side-effect (it is linked to a vertex in the control flow graph), the symbolic state associated with the token changes. For example, let us assume that the evaluation order of the operands is left to right (i.e., in the binary expression a $\oplus$ b, the operand a is first evaluated, then the operand b is evaluated and finally, the result of the operation is computed). The symbolic execution of the expression:

$$*p{+}{+} = {+}{+}b + a{-}{-} + b$$

where the type of a, b and *p is 'int', in the symbolic state:

$$\langle\{\langle a, \alpha\rangle, \langle b, \beta\rangle, \langle p, \chi\rangle, \langle x[0], \gamma\rangle, \langle x[1], \delta\rangle, \ldots\}, PC\rangle$$

where χ and $\chi + 2$ are the symbolic addresses of x[0] and x[1], respectively, is obtained by the following steps:

- evaluate the expression *p referring to the memory location pointed by p and

[7] The operator *val* returns the value of an expression.
[8] Here the operator *val* indicates that the value of the expression b++ has already been computed at a previous control flow vertex.

increment the pointer p; the evaluation result is the memory location val(*p++) = $p(\chi)$ = x[0], while the resulting state is:

$$\langle\{\langle a, \alpha\rangle, \langle b, \beta\rangle, \langle p, \chi{+}2\rangle, \langle x[0], \gamma\rangle, \langle x[1], \delta\rangle, \ldots\}, PC\rangle$$

- increment the value stored in b and evaluate the result; the evaluation result is val(++b) = β+1 and the new state is:

$$\langle\{\langle a, \alpha\rangle, \langle b, \beta{+}1\rangle, \langle p, \chi{+}2\rangle, \langle x[0], \gamma\rangle, \langle x[1], \delta\rangle, \ldots\}, PC\rangle$$

- evaluate the value stored in a and decrement it; the evaluation result will be val(a--) = α while the new state is:

$$\langle\{\langle a, \alpha{-}1\rangle, \langle b, \beta{+}1\rangle, \langle p, \chi{+}2\rangle, \langle x[0], \gamma\rangle, \langle x[1], \delta\rangle, \ldots\}, PC\rangle$$

- evaluate the symbolic expression:

$$\text{val(++b)} + \text{val(a--)} + b = \beta + 1 + \alpha + \beta + 1 = \alpha + 2{*}\beta + 2$$

and assign it to x[0]; the resulting state will be:

$$\langle\{\langle a, \alpha{-}1\rangle, \langle b, \beta{+}1\rangle, \langle p, \chi{+}2\rangle, \langle x[0], \alpha{+}2{*}\beta{+}2\rangle, \langle x[1], \delta\rangle, \ldots\}, PC\rangle$$

Figure 4 shows the CCG representation and the sequence of evaluations and state changing at each node of the syntax tree. The evaluation is made by a depth-first left traversal of the abstract syntax tree of the expression.

A token can generate two different new tokens whenever a predicate vertex is encountered during symbolic execution ('forking' operation) and two or more tokens can be joined into one token by the 'folding' operation. Therefore, tokens advance on the CCG

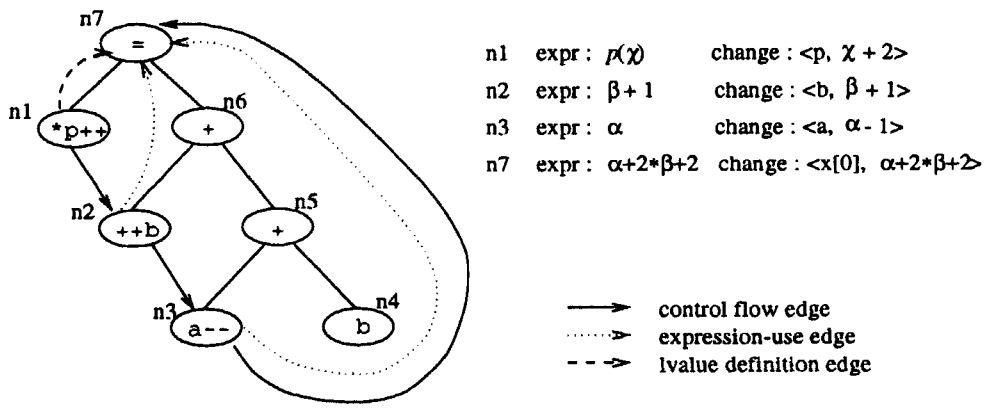


Figure 4. Example CCG subgraph of ambiguous expression and execution sequence

representation in a similar way to concurrent processes. Figure 5 shows the state transition diagram of a token. In particular, one token at a time is *executing*, while tokens which lie on a vertex joining different execution paths are *waiting* and the other tokens are *ready*. The symbolic executor is interactive. The user schedules the tokens for the advancement and provides some assertions to force an execution to follow only one branch of a predicate, whenever the theorem prover is not able to do it. Hence, the user can decide to suspend a token (by making it ready) and execute another one which is in the ready state. Two or more waiting tokens lying on a same vertex, are folded in one token and eventually the user can decide to move them in the ready state. User interaction is also necessary to deal with loops. Whenever the number of times a loop can be executed is not known, symbolic execution fails to keep track of all the paths the loop can generate. This happens when the current path-condition does not imply either the loop condition nor its negation. To correctly handle this problem we need to find a logical formula that is true at the beginning of the loop (before the evaluation of the loop condition) after an arbitrary number $n \geqslant 0$ of iterations. Such a formula is called *loop invariant*. Although in general the problem of finding invariant assertions for a given program is *undecidable*, many methods and tools based both on deterministic and heuristic approaches have been proposed in the past to automatically derive invariants, e.g. (Basu and Misra, 1975; Ellozy, 1981; German and Wegbreit, 1975; Katz and Manna, 1976; Tamir, 1980; Wegbreit, 1974). A more complete discussion and survey can be found in Tamir (1980). These attempts were related to the problem of proving partial program correctness. A different approach by Waters (1979) tries to analyse loops by decomposing them into smaller code fragments and using a library of programming plans to recover loop invariants. A recent variant of this knowledge-based method (Abd-El-Hafiz and Basili, 1993) has been used to mechanically annotate a loop with a formal specification. This approach allows dealing with more complex loops but faces space and time problems to store and search for plans. However, in general, user interaction is required to find and provide an appropriate invariant for a loop in order to continue the symbolic execution of the function.

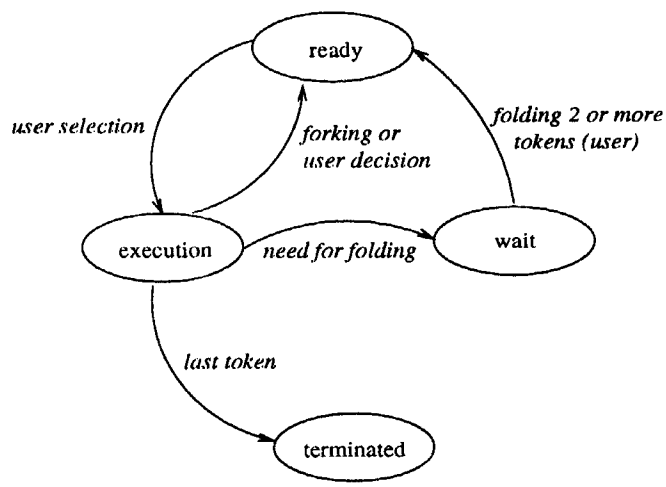


Figure 5. State transition diagram for tokens

5. FINDING SLICING CRITERION AND EXTRACTING SLICES

The precondition of a statement is a first order logic formula which must hold true in order for the statement to be executed. The precondition of a statement can be derived from the symbolic state holding before its execution. For example, let us consider the function absdiff above. From the symbolic state $\langle S_6, P_6 \rangle$ holding before the execution of the statement return diff, the precondition

$$(\alpha - \beta > 0 \wedge \gamma = \alpha - \beta) \vee (\beta - \alpha \geq 0 \wedge \gamma = \beta - \alpha)$$

of the statement can be easily obtained. Each vertex in the control flow subgraph of a program's CCG is therefore annotated with the symbolic state holding before its execution. The slicing criterion finder takes the symbolic state holding before the execution of a statement and the specification of the function to be recovered (together with some user assertions) as input. During symbolic execution the finder first looks for a statement s_{in} whose precondition is equivalent to the precondition of the functional abstraction. Once such a statement has been found, it looks for a statement s_{out} whose precondition is equivalent to the postcondition of the functional abstraction. If also the statement s_{out} is found and s_{in} dominates s_{out}, the slicing criterion $\langle s_{in}, s_{out}, V_{out} \rangle$ is produced, where V_{out} is the set of the program variables corresponding to the output data of the function. Finally, the slice is computed. In the following we will show the specification driven slicing process through an example.

Let us consider the following sample program:

```
main()
{
  int a, b, diff, fact, sum, i;
  scanf("%d", &a);
  scanf("%d", &b);
  if (a > b)
    diff = a - b;
  else
    diff = b - a;
  fact = i = 1;
  sum = 0;
  while (i <= diff) {
    fact *= i;
    sum += i++;
  }
  printf("%d \n", fact);
  printf("%d \n", sum);
}
```

which computes the factorial fact and the sum sum of the absolute value of the difference of two integers a and b. Let us suppose to extract the slice implementing the factorial function, whose specification is:

factorial: $n \in N_0 \rightarrow m \in N$
precondition: *true*
postcondition: $m = n!$

where N is the set of the natural numbers and N_0 is the set of natural numbers including 0. The software engineer must provide some assertions to associate the data of the specification with the variables of the program. In this case he will associate the variable diff with the input data n and the variable fact with the output data m. Moreover, as diff and fact are integer variables while n and m are natural numbers, the specification of the function must be changed according to the program variables in:

factorial: $n \in Z \rightarrow m \in Z$
precondition: $n \geq 0$
postcondition: $m = n!$

where Z is the set of relative numbers. Let us symbolically execute the program above. The symbolic state obtained before the execution of the expression i = 1 is:[9]

$$\langle S_6, P_6 \rangle = \langle \{\langle \text{a}, \alpha \rangle, \langle \text{b}, \beta \rangle, \langle \text{diff}, \gamma \rangle, \langle \text{fact}, \textit{undef} \rangle, \langle \text{sum}, \textit{undef} \rangle, \langle \text{i}, \textit{undef} \rangle\}, (\alpha - \beta > 0 \wedge \gamma = \alpha - \beta) \vee (\beta - \alpha \geq 0 \wedge \gamma = \beta - \alpha) \rangle$$

where α and β are the input symbolic values assigned to a and b, respectively. Then the precondition for i = 1 is:

$$(\alpha - \beta > 0 \wedge \gamma = \alpha - \beta) \vee (\beta - \alpha \geq 0 \wedge \gamma = \beta - \alpha)$$

which is equivalent to the precondition $n \geq 0$ under the assumption $n = \gamma$ (n has been associated with diff and γ is the symbolic value of diff), i.e., $\forall\ \alpha, \beta$

$$(\alpha - \beta > 0 \wedge \gamma = \alpha - \beta) \vee (\beta - \alpha \geq 0 \wedge \gamma = \beta - \alpha) \Rightarrow (n = \gamma \wedge n \geq 0)$$

and $\forall n$

$$(n = \gamma \wedge n \geq 0) \Rightarrow (\alpha - \beta > 0 \wedge \gamma = \alpha - \beta) \vee (\beta - \alpha \geq 0 \wedge \gamma = \beta - \alpha)$$

for each assignment to one of the symbolic values α or β. Therefore, i = 1 is candidate to be the initial statement of the slicing criterion. Before the execution of the while statement, the symbolic state is:

$$\langle S_7, P_7 \rangle = \langle \{\langle \text{a}, \alpha \rangle, \langle \text{b}, \beta \rangle, \langle \text{diff}, \gamma \rangle, \langle \text{fact}, 1 \rangle, \langle \text{sum}, 0 \rangle, \langle \text{i}, 1 \rangle\}, (\alpha - \beta > 0 \wedge \gamma = \alpha - \beta) \vee (\beta - \alpha \geq 0 \wedge \gamma = \beta - \alpha) \rangle$$

At this point, the while loop is encountered. Symbolic execution can be used to obtain the recurrence equations from which the loop invariant is generated (Cimitile, De Lucia

[9] The statement fact = i = 1 gives rise the two CCG vertices (i = 1) and (fact =) in the order.

and Munro, 1995a). The variables which change their state during the execution of the loop are fact, sum and i. The recurrence equations for these variables are:

$$\begin{aligned} \mathsf{fact}_n &= \mathsf{fact}_{n-1} * \mathsf{i}_{n-1} \\ \mathsf{sum}_n &= \mathsf{sum}_{n-1} + \mathsf{i}_{n-1} \\ \mathsf{i}_n &= \mathsf{i}_{n-1} + 1 \\ \mathsf{fact}_0 &= 1 \\ \mathsf{sum}_0 &= 0 \\ \mathsf{i}_0 &= 0 \\ \mathsf{i}_n &\leqslant \gamma \end{aligned}$$

where the values for fact_0, sum_0 and i_0 are obtained from the state $\langle S_7, P_7 \rangle$. The solution for the system above is:

$$\begin{aligned} \mathsf{fact}_n &= n! \\ \mathsf{sum}_n &= n*(n+1)/2 \\ \mathsf{i}_n &= n+1 \\ \mathsf{i}_n &\leqslant \gamma \end{aligned}$$

from which, substituting the value of n obtained from the third equation in the first and second equation and eliminating the subscript n, we obtain the loop-invariant:

$$\begin{aligned} \mathsf{fact} &= (\mathsf{i}-1)! \\ \mathsf{sum} &= \mathsf{i}*(\mathsf{i}-1)/2 \\ \mathsf{i} &\leqslant \gamma \end{aligned}$$

By associating fact, sum and i with the new symbolic constants δ, σ and ι, respectively and executing the loop invariant in the symbolic state $\langle S_7, P_7 \rangle$ we obtain the state at a generic loop iteration:

$$\langle S_8, P_8 \rangle = \langle \{\langle \mathsf{a}, \alpha \rangle, \langle \mathsf{b}, \beta \rangle, \langle \mathsf{diff}, \gamma \rangle, \langle \mathsf{fact}, \delta \rangle, \langle \mathsf{sum}, \sigma \rangle, \langle \mathsf{i}, \iota \rangle\}, ((\alpha-\beta>0 \wedge \gamma = \alpha-\beta) \vee (\beta-\alpha \geqslant 0 \wedge \gamma = \beta-\alpha)) \wedge \delta = (\iota-1)! \wedge \sigma = \iota*(\iota-1)/2 \wedge \iota \leqslant \gamma \rangle$$

At this point, let us suppose that the condition of the loop is false (i.e. $\iota = \gamma+1$) and the loop cannot be further executed. The symbolic state at the exit of the loop will be:

$$\langle S_9, P_9 \rangle = \langle \{\langle \mathsf{a}, \alpha \rangle, \langle \mathsf{b}, \beta \rangle, \langle \mathsf{diff}, \gamma \rangle, \langle \mathsf{fact}, \delta \rangle, \langle \mathsf{sum}, \sigma \rangle, \langle \mathsf{i}, \iota \rangle\}, ((\alpha-\beta > 0 \wedge \gamma = \alpha-\beta) \vee (\beta-\alpha \geqslant 0 \wedge \gamma = \beta-\alpha)) \wedge \delta = (\iota-1)! \wedge \sigma = \iota*(\iota-1)/2 \wedge \iota = \gamma+1 \rangle$$

This symbolic state holds before the execution of the statement printf("%d \n", fact) whose precondition is then:

$$((\alpha-\beta > 0 \wedge \gamma = \alpha-\beta) \vee (\beta-\alpha \geqslant 0 \wedge \gamma = \beta-\alpha)) \wedge \iota = \gamma+1 \wedge \delta = (\iota-1)! \wedge \sigma =$$

$$\iota*(\iota-1)/2$$

which can also be written as:

$$((\alpha-\beta > 0 \wedge \gamma = \alpha-\beta) \vee (\beta-\alpha \geqslant 0 \wedge \gamma = \beta-\alpha)) \wedge \iota = \gamma+1 \wedge \delta = \gamma! \wedge \sigma = \gamma*(\gamma+1)/2$$

It is easy to prove, in a similar way as before, that whenever the precondition of the specification of the function *factorial* holds true, the condition above is equivalent to the postcondition of the specification. Moreover, as the statement i = 1 dominates the statement printf("%d \n", fact) the slicing criterion ⟨i = 1, printf("%d \n", fact), fact⟩ is produced. The slice on this slicing criterion can be easily extracted using algorithms exploiting control flow edges and control and data dependencies of the CCG, on the same line as the algorithms proposed in Cimitile and De Lucia, 1995; Ottenstein and Ottenstein, 1984; Horwitz, Reps and Binkley, 1990; Weiser, 1984). The slice produced is:

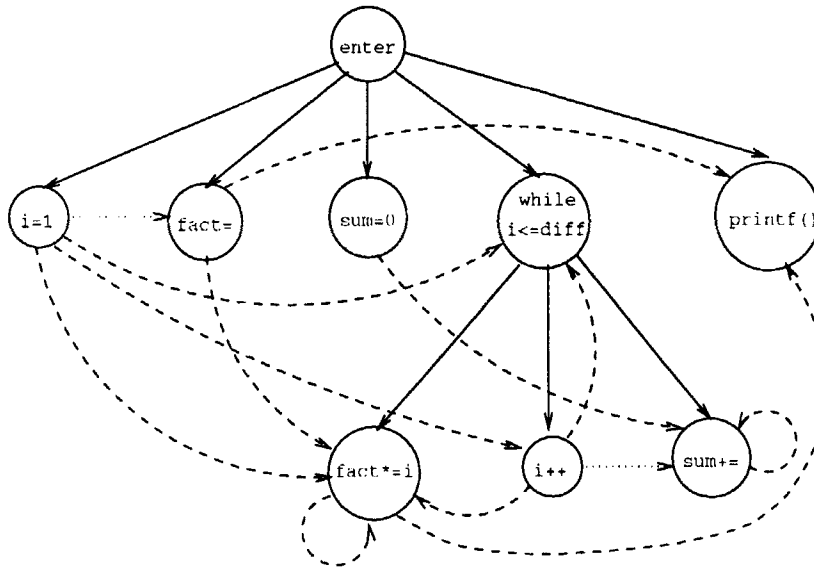
```
fact = i = 1;
while (i <= diff) {
  fact *= i;
  i++;
}
```

Note that in the statement sum += i++ only the expression i++ is considered in the slice. This accuracy is possible thanks to the extra vertices representing embedded side-effects in the CCG (Kinloch and Munro, 1994). Figure 6(a) shows a partial view of the CCG corresponding to the code fragment:

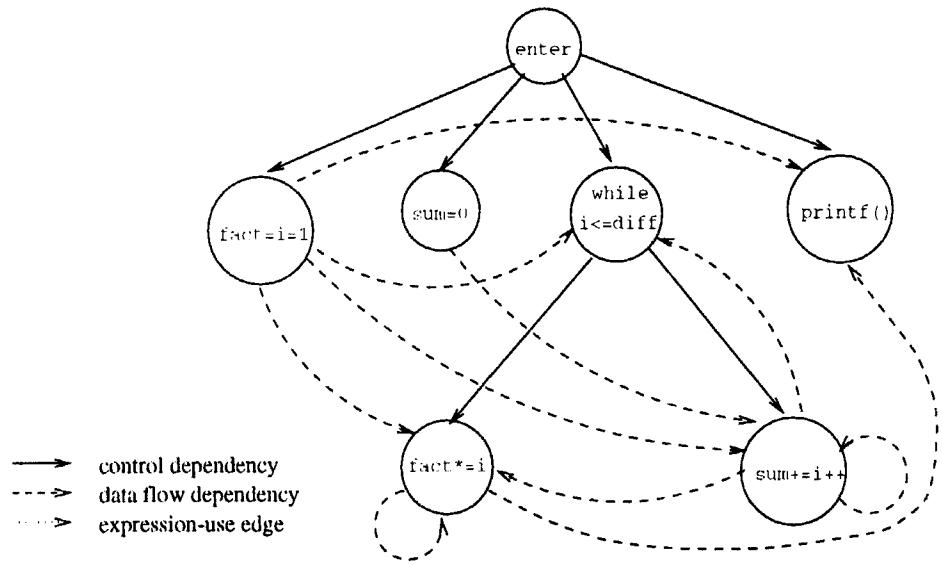
```
fact = i = 1;
sum = 0;
whlie (i <= diff) {
  fact *= i;
  sum += i++;
}
printf("%d \n", fact);
```

Control and data dependence edges and expression-use edges are represented in the figure. If no extra nodes are created for the embedded side-effects i = 1 and i++ the graph in Figure 6(b) results and the program slice becomes:

```
fact = i = 1;
sum = 0;
while (i <= diff) {
  fact *= i;
  sum += i++;
}
```



(a): Refined CCG



(b): Embedded side-effects

Figure 6. Enhanced slicing accuracy with refined CCG

6. RELATED WORK

Several specification driven methods for identifying abstractions in code are knowledge-based approaches. They encode the knowledge about the functions to be identified in the form of programming plans and can be classified as either top-down or bottom-up methods. Top-down methods (Kontagiannis *et al.*, 1994; Kozaczynsky, ning and Engberts, 1992, 1994) use the knowledge about the goals the program is assumed to achieve and some heuristics to locate both the program statements and the plan from the library that can achieve these goals; then they exploit pattern matching techniques to map the statements to the plan. The accuracy of such a method depends on the complexity of the heuristics. Bottom-up methods (Abd-El-Hafiz and Basili, 1993; Quilici, 1994; Wills, 1990, 1992) are more precise but also more expensive. They start from the program statements and try to identify the plan which can have these statements as components; then they attempt to infer program goals from these plans. Such methods are too expensive because they exhaustively search for plans in a program. For example, Wills (1990, 1992) encodes the program into a flow-graph and uses flow-graph grammar as a library of clichés. Graph parsing is then exploited to identify all the notable subgraphs in the program's flow-graph. The complexity time of this approach is exponential because of the NP-completeness of the problem. A solution to limit the number of candidate plans considered during program understanding can be provided by using indexed or hierarchical organizations of the plan library (Abd-El-Hafiz and Basili, 1993; Quilici, 1994). However, the main limitation of knowledge-based methods is that they can require a large library of plans. Moreover, while this approach can be effective for recognizing stereotypical domain independent plans, it can be too expensive for dealing with domain dependent functions and then not convenient for a reuse re-engineering process. Indeed, to apply a knowledge-based method to software written for a particular application domain we need to design and develop a new plan library for this domain and the cost of such a task might be comparable to the cost of designing and developing a library of reusable modules.

A cost-effective approach for locating functionalities in code has been proposed by Wilde *et al.*, 1992; Wilde and Scully, 1995. The method uses well defined test cases as specifications of the function to be extracted. The program is then instrumented so that the code components exercised by the test cases can be identified. Both a deterministic and probabilistic techniques have been proposed to analyse the traces resulting from the program execution. While this method is very practical and easy to implement and use, it is only good to find components that are unique to a particular functionality. In general, the method lacks precision, because the software component identified could be too large and include more functionalities than the one sought. In some cases, the set of statements exercised by a set of test cases might include the whole program. A more precise method (Hall, 1995) combines the use of a set of test cases with program slicing. The method is called *simultaneous dynamic program slicing* because it extends and simultaneously applies to a set of test cases *dynamic slicing* techniques (Korel and Laski, 1990), which produce executable slices that are correct on only one input. This method takes into account the data flow of the program and then allows the reduction of the set of selected statements. Indeed, only the statements that might affect the values of the output variables of the function on the exercised paths will be considered. However, the method does not consider the problem of finding a slicing criterion. This can result in lack of precision when dealing with large programs. Whenever a slicing criterion has not been adequately

selected, the method might produce slices containing more functionalities than the one expected or even not containing all the statements of the searched functional abstraction. The use of a formal specification and symbolic execution helps to define a suitable slicing criterion, making the specification driven program slicing more precise in the identification of expected functions.

Formal methods techniques have also been used in other fields of software maintenance. For example, the authors used symbolic execution for abstracting formal specifications (in the form of preconditions and postconditions) from code of modules implementing functional abstractions (Cimitile, De Lucia and Munro, 1995a). Symbolic execution and partial evaluation have also been used to specialize programs for specific values of their input variables (Coen-Porisini *et al.*, 1991; Blazy and Facon, 1994). The simplified programs behave like the initial ones for the specific values. This technique is useful both for software reuse (Coen-Porisini *et al.*, 1991) and program comprehension (Blazy and Facon, 1994).

7. CONCLUSIONS

We presented a specification driven slicing process for recovering functional abstractions from code. The process combines program slicing and symbolic execution techniques. We use a new definition of a program slice which contains two statements in the slicing criterion. The two statements delimit the region of code in which the slice must be computed. In this way we can obtain more precise slices with respect to the functional abstraction to be recovered. The slice extracted can be easily re-engineered and clustered into a module. The specification of the function to be isolated is used together with symbolic execution and theorem proving techniques to correctly identify the initial and final statements of the slicing criterion. Symbolic execution allows the association of a program statement or predicate with its precondition, i.e. the condition which must hold on the program variables before its execution. The specification of the functional abstraction, given in terms of a precondition and a postcondition, is then compared with the conditions associated with program statements, also called invariant assertions. The statement whose precondition is equivalent to the precondition of the functional abstraction is candidate to be the initial statement of the slicing criterion, while the statement whose precondition is equivalent to the postcondition of the functional abstraction is candidate to be the final statement of the slicing criterion. The slicing criterion is produced whenever the initial statement dominates the final statement on the control flow. Human interaction is required during this task. First, the software engineer must associate the output data of the specification with the program's variables. Moreover, as the problem of finding invariant assertions is, in general, *undecidable*, symbolic execution can require human interaction in order to prove some implications and assert some invariants.

The specification driven program slicing process is language independent. However, we are currently developing the process for programs written in C language. A fine-grained representation for C programs, the Combined C Graph (CCG), is produced by a program analyser. The CCG contains the features of several different program representations and can be used for most of the software maintenance tasks, allowing a better integration of different software tools. The CCG analyser (Kinloch and Munro, 1994) produces a data base of Prolog (Sterling and Shapiro, 1986) facts and is composed of two subsystems. The first subsystem is the PERPLEX tool (Bunter, 1993) which is written using the

YACC (Johnson, 1975) compiler-compiler and uses grammar corresponding to that in Kernighan and Ritchie (1988). PERPLEX produces a Prolog database containing several types of facts that support intraprocedural information about the individual FCCGs. The second subsystem consists of three analysis meta programs (Kinloch and Munro, 1994) written in Prolog that enrich the database produced by PERPLEX with interprocedural information, and control and data dependencies.

A program slicer has also been implemented in Prolog by adapting the slicing algorithm proposed in Cimitile and De Lucia (1995) to the Combined C Graph. The program slicer takes a slicing criterion of the type $\langle s_{in}, s_{out}, V_{out} \rangle$ as defined in Section 2, where V_{out} is the set of variables referenced as s_{out}, as input. The slice is computed by backwards traversing CCG dependence edges. The algorithm implemented works both at intraprocedural and interprocedural level. At intraprocedural level, it only considers in the slice the set Pathlist of CCG nodes lying on a control flow path between the statements s_{in} and s_{out} of the slicing criterion. At interprocedural level, the algorithm only considers the set Funlist of functions that can be reached through a call chain from a call site contained in Pathlist. In this way only interprocedural control flow paths between s_{in} and s_{out} can be considered.

The implementation of a Prolog prototype for symbolic execution and slicing criterion finding is in progress. This subsystem is composed of four Prolog modules (*scheduler*, *executor*, *evaluator*, and theorem_prover) that are loaded into the Prolog environment together with the CCG fact base and the specification of the functional abstraction. The scheduler is in charge to select the token (and then the execution path) to be advanced. Human interaction can be required to make some decisions. When a token has been selected for the execution, the statement corresponding to the token is executed. Different cases are considered for the execution of loops, predicates, function calls and function returns. The executor invokes the evaluator whenever an expression has to be evaluated. The CCG abstract syntax tree corresponding to the expression is then traversed and a logic formula is produced. The logic formulas in a symbolic state are expressed as Prolog terms and are universally quantified as well as the specification of the function. The theorem_prover is invoked whenever an implication must be proved. In particular, a proof is required each time a predicate is encountered during symbolic execution. Moreover, after each token execution, the user can invoke the module to prove the implication between the current path condition and the specification of the function, in order to find a slicing criterion.

Although until now program slicing has been used to isolate reusable functions in large monolithic programs written in COBOL language, we are confident that this approach can give good results also for programs written in C language. Indeed, even though the C language provides a primitive for implementing functional abstractions, very often a single C *function* implements more than one functional abstraction. We conducted a case study (Cimitile, De Lucia and Munro, 1995b) which validated this assertion. The program chosen for the experiment is PERPLEX (Bunter, 1993), a medium sized system. PERPLEX underwent two perfective maintenance interventions that increased its size by more than 2 000 LOC. The version used for the experiment was over 5 000 LOC long (not including comments) (Cimitile, De Lucia and Munro, 1995b). Four functions between 168 and 392 LOC long were selected for the experiment. Using the specification driven program slicing we were able to decompose these functions into 18 subfunctions. After the isolation phase, the subfunctions identified were re-engineered. In the final version the new functions were

between 23 and 90 LOC long. Only one function was 150 LOC long: we identified further subfunctions in this function, but did not decompose it because we used 150 LOC as a threshold for the length of a function.

The total effort for identifying the functions was seven hours. The preconditions of the functions were automatically identified. Symbolic execution and theorem proving required intensive user interactions for the identification of the postconditions due to the presence of dynamic variables and function calls. However, symbolic execution and theorem proving are an interactive support which help the maintainer during the manipulation of the expressions.

The effort required for the comprehension process during the maintenance of the system has been reduced and the reusability of the functions has been improved as a result of the re-engineering process. Moreover, the case study demonstrated that even though the original design has been made following software engineering principles, maintenance operations (in particular perfective maintenance) can add new functionalities (in terms of code) to existing functions and sometimes the same code in different functions. Duplicated functionalities were recovered using the specification driven slicing process, re-engineered and clustered into single C *functions* (Cimitile, De Lucia and Munro, 1995b).

We are currently investigating the possibility of scaling up this method to larger programs. In a first analysis, problems might arise due to the size of both the CCG and the expressions manipulated by the symbolic executor and the theorem prover. In this case, other structural modularization techniques can be used to decompose the system into smaller modules (Canfora, Cimitile and Visaggio, 1995). Each module can then be searched for specific functional abstractions using specification driven program slicing.

The work described in this paper is part of RE^2 (Canfora, Cimitile and Munro, 1994), a research project addressing software reuse jointly carried out by DIS (Department of 'Informatica e Sistemistica') at the University of Naples and CSM (Centre for Software Maintenance) at the University of Durham. The project started in 1992 and has been partially funded by CNR (Italian National Research Council) within the wider national project 'Sistemi Informatici e Calcolo Parallelo'. The RE^2 project outlines the role that reverse engineering and re-engineering techniques have in *reuse re-engineering* processes aiming at identifying, extracting and re-engineering software components from existing systems that can be reused in the development of new software projects. The main results of the RE^2 project are the definition and implementation of severral code scavenging techniques for the identification of different types of abstractions (Canfora, Cimitile and Visaggio, 1995). We are currently transferring methods, technologies and tools developed and used in academic laboratories to industrial scale projects.

References

Abd-El-Hafiz, S. K. and Basili, V. R. (1993) 'Documenting programs using a library of tree structured plans', in *Proceedings of the International Conference on Software Maintenance*, Montreal, Canada, IEEE Comp. Soc. Press, Washington, DC, pp. 152–161.

Abd-El-Hafiz, S. K., Basili, V. R. and Caldiera, G. (1991) 'Towards automated support for extraction of reusable components', in *Proceedings of the International Conference on Software Maintenance*, Sorrento, Italy, IEEE Comp. Soc. Press, pp. 212–219.

Aho, A. V., Sethi, R. and Ullmann, J. D. (1986) *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, MA.

Andrews, P. B. (1981) 'Theorem proving via general matings', *Journal of the ACM*, **28**(2), 193–214.

Arnold, R. S. and Frakes, W. B. (1992) 'Software reuse and reengineering', *CASE Trends*, **4**(2), 44–48.

Basili, V. R. and Caldiera, G. (1991) 'Identifying and qualifying reusable software components', *IEEE Computer*, **24**(2), 61–70.

Basu, S. K. and Misra, J. (1975) 'Proving loop programs', *IEEE Transactions on Software Engineering*, **SE-1**(1), 76–86.

Bennett, K. (1995) 'Legacy systems: coping with success', *IEEE Software*, **12**(1), 19–23.

Biggerstaff, T. J., Mitbander, B. G. and Webster, D. (1994) 'Program understanding and the concept assignment problem', *Communications of the ACM*, **37**(5), 72–83.

Blazy, S. and Facon, P. (1994) 'SFAC, a tool for program comprehension by specialization', in *Proceedings of the 3rd Workshop on Program Comprehension*, Washington, DC, IEEE Comp. Soc. Press, Washington, DC, pp. 162–167.

Boyer, R. S. and Moore, J. S. (1979) *A Computational Logic*, Academic Press, New York.

Bunter, T. (1993) 'PERPLEX: an extensible tool architecture for C source code', Technical Report Computer Science 6/93, School of Engineering and Computer Science, University of Durham.

Callahan, D. (1988) 'The program summary graph and flow-sensitive interprocedural data flow analysis', in *Proceedings of the SIGPLAN Conference on Programming Languages Design and Implementation*, Atlanta, Georgia, pp. 47–56.

Canfora, G., Cimitile, A. and Munro, M. (1994) 'RE^2: reverse engineering and reuse re-engineering', *Journal of Software Maintenance: Research and Practice*, **6**(2), 53–72.

Canfora, G., Cimitile, A., De Lucia, A. and Di Lucca, G. A. (1994a) 'Software salvaging based on conditions', in *Proceedings of the International Conference on Software Maintenance*, Victoria, Canada, IEEE Comp. Soc. Press, Washington, DC, pp. 424–433.

Canfora, G., De Lucia, A., Di Lucca, G. A. and Fasolino, A. R. (1994b) 'Slicing large programs to isolate reusable functions', in *Proceedings of the EUROMICRO Conference*, Liverpool, U.K., IEEE Comp. Soc. Press, Washington, DC, pp. 140–147.

Canfora, G., Cimitile, A. and Visaggio, G. (1995) 'Assessing modularization and code scavenging techniques', *Journal of Software Maintenance: Research and Practice*, **7**(5), 317–331.

Chikofsky, E. J. and Cross II, J. H. (1990) 'Reverse engineering and design recovery: a taxonomy', *IEEE Software*, **7**(1), 13–17.

Cimitile, A. and De Lucia, A. (1995) 'Isolating functional abstractions in existing code for software reuse', Technical Report CNR PF-SICP no. 6/119, Department of 'Informatica e Sistemistica', University of Naples.

Cimitile, A., De Lucia, A. and Munro, M. (1995a) 'Qualifying reusable functions using symbolic execution', in *Proceedings of the 2nd Working Conference on Reverse Engineering*, Toronto, Canada, IEEE Comp. Soc. Press, Washington, DC, pp. 178–187.

Cimitile, A., De Lucia, A. and Munro, M. (1995b) 'Identifying reusable functions using specification driven program slicing: a case study', in *Proceedings of the International Conference on Software Maintenance*, Opio (Nice), France, IEEE Comp. Soc. Press, Washington, DC, pp. 124–133.

Cimitile, A., Fasolino, A. R. and Maresca, P., (1993) "Reuse-Reengineering and Validation via Concept Assignment", in *Proceedings of the International Conference on Software Maintenance*, Montreal, Canada, IEEE Comp. Soc. Press, Washington, DC, pp. 216–225.

Cimitile, A. and Visaggio, G. (1995) 'Software salvaging and the call dominance tree', *The Journal of Systems and Software*, **28**(2), 117–127.

Clarke, L. A. and Richardson, D. J. (1981) 'Symbolic evaluation methods—implementations and applications', in Chandrasekaran, B. and Radicchi, S. (Eds), *Computer Program Testing*, North-Holland, Amsterdam, pp. 65–102.

Coen-Porisini, A., De Paoli, F., Ghezzi, C. and Mandrioli, D. (1991) 'Software specialization via symbolic execution', *IEEE Transactions on Software Engineering*, **17**(9), 884–899.

Cooper, K. and Kennedy, K. (1989) 'Fast interprocedural alias analysis', in *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, Austin, Texas, ACM Press, New York, pp. 49–59.

Coward, P. D. (1988) 'Symbolic execution systems—a review', *Software Engineering Journal*, **3**(6), 229–239.

Dannenberg, R. B. and Ernst, G. W. (1982) 'Formal program verification using symbolic execution', *IEEE Transactions on Software Engineering*, **SE-8**(1), 43–52.

De Lucia, A., Imperatore, A., Napoli, M., Tortora, G. and Tucci, M. (1992) 'The software development workbench WSDW', in *Proceedings of the 4th International Conference on Software Engineering and Knowledge Engineering*, Capri, Italy, IEEE Comp. Soc. Press, Washington, DC, pp. 213–221.

Ehrig, H., Kreowski, H. J., Maggiolo Schettini, A., Rosen, B. K. and Winkowski, J. (1981) 'Transformations of structures: an algebraic approach', *Math. Syst. Theory*, **14**, 305–334.

Ellozy, H. A. (1981) 'The determination of loop invariants for programs with arrays', *IEEE Transactions on Software Engineering*, **SE-7** (2), 197–206.

Fenton, N. E. (1991) *Software Metrics: A Rigorous Approach*, Chapman & Hall, London.

Ferrante, J., Ottenstein, K. J. and Warren, J. (1987) 'The program dependence graph and its use in optimization', *ACM Transactions on Programming Languages and Systems*, **9**(3), 319–349.

Gallagher, K. B. and Lyle, J. R. (1991) 'Using program slicing in software maintenance', *IEEE Transactions on Software Engineering*, **17**(8), 751–761.

German, M. and Wegbreit, B. (1975) 'A synthesizer of inductive assertions', *IEEE Transactions on Software Engineering*, **SE-1** (1), 68–75.

Hall, R. J. (1995) 'Automatic extraction of executable program subsets by simultaneous program slicing', *Automated Software Engineering*, **2**(1), 33–53.

Hantler, S. L. and King, J. C. (1976) 'An introduction to proving the correctness of programs', *Computing Surveys*, **8**, 331–353.

Harrold, M. J. and Malloy, B. (1993) 'A unified interprocedural program representation for a maintenance environment', *IEEE Transactions on Software Engineering*, **19**(6), 584–593.

Harrold, M. J. and Soffa, M. L. (1990) 'Computation of interprocedural definition and use dependencies', in *Proceedings of the IEEE International Conference on Computer Languages*, New Orleans, Louisiana, IEEE Comp. Soc. Press, Washington, DC, pp. 297–306.

Harrold, M. J. and Soffa, M. L. (1991) 'Selecting data-flow integration testing', *IEEE Software*, **8**(2), 58–65.

Hecht, M. S. (1977) *Flow Analysis of Computer Programs*, Elsevier North-Holland, New York.

Hoare, C. A. R. (1969) 'An axiomatic basis for computer programming', *Communications of the ACM*, **12**(10), 576–580.

Horwitz, S., Reps, T. and Binkley, D. (1990) 'Interprocedural slicing using dependence graphs', *ACM Transactions on Programming Languages and Systems*, **12**(1), 26–60.

Johnson, S. C. (1975) 'YACC: yet another compiler-compiler', Technical Report 32, Bell Laboratories, Murray Hills, New Jersey. Also in *UNIX Programmers' Guide*.

Katz, S. and Manna, Z. (1976) 'Logical analysis of programs', *Communications of the ACM*, **19**(4), 188–206.

Kernighan, B. and Ritchie, D. (1988) *The C Programming Language*, 2nd edn, Prentice-Hall, Englewood Cliffs, New Jersey.

King, J. C. (1976) 'Symbolic execution and program testing', *Communications of the ACM*, **19**(7), 385–394.

Kinloch, D. A. and Munro, M. (1994) 'Understanding C programs using the combined C graph representation', in *Proceedings of the International Conference on Software Maintenance*, Victoria, Canada, IEEE Comp. Soc. Press, Washington, DC, pp. 172–180.

Kontogiannis, K., De Mori, R., Bernstein, M. and Merlo, E. (1994) 'Localization of design concepts in legacy systems', in *Proceedings of the International Conference on Software Maintenance*, Victoria, Canada, IEEE Comp. Soc. Press, Washington, DC, pp. 414–423.

Korel, B. and Laski, J. (1990) 'Dynamic slicing of computer programs', *The Journal of Systems and Software*, **13**(3), 187–195.

Kozaczynsky, W., Ning, J. and Engberts, A. (1992) 'Program concept recognition and transformation', *IEEE Transactions on Software Engineering*, **18**(12), 1065–1075.

Kozaczynsky, W., Ning, J. and Engberts, A. (1994) 'Automated program understanding by concept recognition', *Automated Software Engineering*, **1**(1), 61–78.

Kuck, D. J. (1978) *The Structure of Computers and Computations*, Wiley, New York.

Landi, W. and Ryder, B. G. (1991) 'Pointer-induced aliasing: a problem classification', in *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages*, Orlando, Florida, ACM Press, New York, pp. 93–103.

Lanubile, F. and Visaggio, G. (1993) 'Function recovery based on program slicing', in *Proceedings*

of the International Conference on Software Maintenance, Montreal, Canada, IEEE Comp. Soc. Press, Washington, DC, pp. 396–404.

Lehman, M. M. (1984) 'Program evolution', *Information Processing Management*, **20**, 19–36.

Lim, W. C. (1994) 'Effects of reuse on quality, productivity and economics', *IEEE Software* **11**(5), 23–30.

Livadas, P. E. and Croll, S. (1994) 'System dependence graphs based on parse tree and their use in software maintenance', *Journal of Information Sciences*, **76**, 197–232.

Maggiolo Schettini, A., Napoli, M. and Tortora, G. (1988) 'Web structures: a tool for representing and manipulating programs', *IEEE Transactions on Software Engineering*, **14**(11), 1597–1609.

Myers, E. M. (1981) 'A precise interprocedural data flow algorithm', in *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Language*, ACM Press, New York, pp. 219–230.

Ning, J. Q., Engberts, A. and Kozaczynski, W. (1993) 'Recovering reusable components from legacy systems by program segmentation', in *Proceedings of the Working Conference on Reverse Engineering*, Baltimore, Maryland, IEEE Comp. Soc. Press, Washington, DC, pp. 64–72.

Ottenstein, K. J. and Ottenstein, L. M. (1984) 'The program dependence graph in a software development environment', in *Proceedings of the SIGPLAN/SIGSOFT Symposium on Practical Programming Development Environments*, Pittsburgh, PA, *ACM SIGPLAN Notices*, **19**(5), 177–184, and *Software Engineering Notes*, **9**(3).

Pande, H. D., Landi, W. A. and Ryder, B. G. (1994) 'Interprocedural def-use associations for C systems with single level pointers', *IEEE Transactions on Software Engineering*, **20**(5), 385–403.

Paulson, L. C. (1982) 'The foundation of a generic theorem prover', *Journal of Automatic Reasoning*, **5**(3), 363–398.

Quilici, A. (1994) 'A memory-based approach to recognizing programming plans', *Communications of the ACM*, **37**(5), 84–93.

Rich, C. and Waters, R. (1990) *The Programmer's Apprentice*, Addison Wesley, Reading, MA.

Ryder, B. G. (1979) 'Constructing the call graph of a program', *IEEE Transactions on Software Engineering*, **SE-5**(3), 216–225.

Sneed, H. M. (1995) 'Planning the reengineering of legacy systems', *IEEE Software*, **12**(1), 24–34.

Sneed, H. M. and Nyary, E. (1994) 'Downsizing large application programs', *Journal of Software Maintenance: Research and Practice*, **6**(5), 105–116.

Soloway, E. and Erdlich, K. (1984) 'Empirical studies of programming knowledge', *IEEE Transactions on Software Engineering*, **SE-10**(5), 595–609.

Sterling, L. and Shapiro, E. (1986) *The Art of Prolog*, MIT Press, Cambridge, MA.

Tamir, M. (1980) 'ADI: automatic derivation of invariants', *IEEE Transactions on Software Engineering*, **SE-6**(1), 40–48.

Tip, F. (1995) 'A survey of program slicing techniques', *Journal of Programming Languages*, **3**, 121–189.

Torczan, L. and Kennedy, K. (1984) 'Efficient computation of flow insensitive interprocedural summary information', in *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, Montreal, Canada, *ACM SIGPLAN Notices*, **19**(6), 49–59.

Waters, R. C. (1979) 'A method for analyzing loop programs', *IEEE Transactions on Software Engineering*, **SE-5**(3), 237–247.

Wegbreit, B. (1974) 'The synthesis of loop predicates', *Communications of the ACM*, **17**(2), 102–112.

Weiser, M. (1981) 'Program slicing', in *Proceedings of the 5th International Conference on Software Engineering*, San Diego, CA, IEEE Comp. Soc. Press, Washington, DC, pp. 439–449.

Weiser, M. (1982) 'Programmers use slices when debugging', *Communications of the ACM*, **25**, 446–452.

Weiser, M. (1984) 'Program slicing', *IEEE Transactions on Software Engineering*, **SE-10**(4), 352–357.

Wilde, N., Gomez, J. A., Gust, T. and Strasburg, D. (1992) 'Locating user functionality in old code', in *Proceedings of the International Conference on Software Maintenance*, Orlando, Florida, IEEE Comp. Soc. Press, Washington, DC, pp. 200–205.

Wilde, N. and Scully, M. C. (1995) 'Software reconnaissance: mapping program features to code', *Journal of Software Maintenance: Research and Practice*, **7**(1), 49–62.

Wills, L. (1990) 'Automated program recognition: a feasibility demonstration', *Artificial Intelligence*, **45**(1–2), 113–171.
Wills, L. (1992) 'Automated program recognition by graph parsing', Ph.D. Thesis, MIT, Cambridge, MA.

Author Biographies:

Aniello Cimitile received the Laurea degree in Electronic Engineering from the University of Naples, Italy, in 1973. He is currently a full Professor of Computer Science at the University of Salerno. Previously, he was with the Department of 'Informatica e Sistemistica' at the University of Naples. Since 1973 he has been a researcher in the field of software engineering and his list of publications contains more than 80 papers published in journals and conference proceedings. He serves in the program and organising committees of several international conferences in the fields of software engineering and software maintenance.
His research interests include software maintenance and testing, software quality, reverse engineering and reuse-reengineering.

Andrea De Lucia was born in Caserta, Italy, in 1968. He received the Laurea degree in Computer Science from the University of Salerno, Italy, in 1991 and the M.Sc. in Computer Science from the University of Durham, U.K., in 1995. He is currently a Ph.D. candidate in Electronic Engineering and Computer Science at the Department of 'Informatica e Sistemistica' of the University of Naples, Italy.
His research interests include software maintenance, reverse engineering, program comprehension, and reuse-reengineering.

Malcolm Munro is Senior Lecturer in Computer Science at the University of Durham, U.K. He is a founder member of The Centre for Software Maintenance and has been an active researcher in software maintenance since 1987. Particular research topics are program comprehension, software visualisation, reverse engineering, and reuse-reengineering. He has served on the program and organising committees of a number of international conferences.